

Appendix C

Selected Exercise Solutions

C.1 Problem 1.8.1 : Checking trigonometric identities

The Fortran 90 program and output follow. The error levels are due to the fact that F90 defaults to single precision reals. F90 is easily extended to double precision, and in theory supports any level of user specified precision. For simplicity the F77 default naming convention for integers and reals is used. That is not a good practice since safety dictates declaring the type of each variable at the beginning of each program. (Try changing the reals to double precision to verify that the error is indeed reduced.)

```
[ 1] implicit none
[ 2] integer :: k,n = 16
[ 3] real, parameter :: pi = 3.141592654 ! set constant
[ 4] print *, ' Theta      sin^2+cos^2      error'
[ 4] do k = 0, n           ! Loop over (n+1) points
[ 5]   theta = k*pi/n
[ 6]   sint = sin( theta )
[ 7]   cost = cos( theta )
[ 8]   test = sint*sint + cost*cost
[ 9]   write (*, '( 3(1pe14.5) )') theta, test, 1.-test
[10] end do ! over k
```

| Theta | sin^2+cos^2 | error |
|-------------|-------------|-------------|
| 0.00000E+00 | 1.00000E+00 | 0.00000E+00 |
| 1.96350E-01 | 1.00000E+00 | 5.96046E-08 |
| 3.92699E-01 | 1.00000E+00 | 0.00000E+00 |
| 5.89049E-01 | 1.00000E+00 | 0.00000E+00 |
| 7.85398E-01 | 1.00000E+00 | 5.96046E-08 |
| 9.81748E-01 | 1.00000E+00 | 0.00000E+00 |
| 1.17810E+00 | 1.00000E+00 | 5.96046E-08 |
| 1.37445E+00 | 1.00000E+00 | 0.00000E+00 |
| 1.57080E+00 | 1.00000E+00 | 0.00000E+00 |
| 1.76715E+00 | 1.00000E+00 | 5.96046E-08 |
| 1.96350E+00 | 1.00000E+00 | 0.00000E+00 |
| 2.15985E+00 | 1.00000E+00 | 0.00000E+00 |
| 2.35619E+00 | 1.00000E+00 | 5.96046E-08 |
| 2.55254E+00 | 1.00000E+00 | 0.00000E+00 |
| 2.74889E+00 | 1.00000E+00 | 0.00000E+00 |
| 2.94524E+00 | 1.00000E+00 | 0.00000E+00 |
| 3.14159E+00 | 1.00000E+00 | 0.00000E+00 |

C.2 Problem 1.8.2 : Newton-Raphson algorithm

The most convenient form of loop is the post-test loop, which allows each iteration to be calculated and the error checked at the end.

```
xnew = x
do {
  x = xnew
  xnew = x - f(x)/fprime(x)
}
while (abs(xnew-x) < tolerance)
```

The alternate logic constructs employ tests at the end of the loop and transfer out the end of the loop when necessary. MATLAB and C++ transfer using the “break” command while F90 uses the “exit” command.

A F90 program with an infinite loop, named `testnewton.f90`, and its result is given below. Be warned that this version uses the `IMPLICIT` name styles for integers and reals instead of the better strong typing that results from the recommended use of `IMPLICIT NONE`.

```
[ 1] function f(x) result(y)
[ 2]   real, intent (in) :: x
[ 3]   real               :: y
[ 4]   y = exp(2*x) - 5*x - 1
[ 5] end function f
[ 6] !
[ 7] function fprime(x) result(y)
[ 8]   real, intent (in) :: x
[ 9]   real               :: y
[10]   y = 2*exp(2*x) - 5
[11] end function fprime
[12] !
[13] program main
[14] implicit none
[15] real, parameter :: tolerance = 1.e-6 ! set constant
[16] real :: x, xnew = 3. ! Initial value
[17] integer :: iteration
[18]   iteration = 0
[19] ! Iteration count
[20] do ! forever until true
[21]   iteration = iteration + 1
[22]   x         = xnew
[23]   xnew      = x - f(x)/fprime(x)
[24]   if ( abs(xnew - x) < tolerance ) exit ! converged is true
[25] end do ! forever
[26] print *, 'Solution: ', xnew, ', Iterations:', iteration
[27] end program main

>>f90 -o newton testnewton.f90
>>newton
Solution:    0.8093941 , Iterations: 10
```

C.3 Problem 1.8.3 : Game of life

```
[ 1] program game_of_life ! procedural version
[ 2] implicit none
[ 3] integer, parameter :: boardsize = 10
[ 4] integer            :: board (boardsize, boardsize) = 0
[ 5] integer            :: newboard (boardsize, boardsize)
[ 6] character(len=1)  :: ok ! page prompt
[ 7] integer            :: k, number ! loops
[ 8]
[ 9] ! Initial life data, the "Glider"
[10] board (3, 3) = 1; board (4, 4) = 1; board (5, 4) = 1
[11] board (5, 3) = 1; board (5, 2) = 1
[12]
[13] print *, "Initial Life Display:"
[14] call spy (board) ! show initial lifeforms
[15] print *, "Initially alive = ", sum (board); print *, " "
[16]
[17] print *, "Enter number of generations to display:"
[18] read *, number
[19] do k = 1, number
[20]   newboard = next_generation (board)
[21]   board    = newboard ! save current lifeforms
[22]   call spy (board) ! show current lifeforms
[23]   print * ; print *, "Generation number = ", k
[24]   print *, "Currently alive = ", sum (newboard)
[25]
[26]   print *, 'continue? (y, n)'
[27]   read *, ok ! read any character to continue
[28]   if ( ok == 'n' ) exit ! this do loop only
[29] end do ! on k for number of generations
[30]
[31] contains ! internal (vs external) subprograms
[32]
[33] function next_generation (board) result (newboard)
[34] ! Compute the next generation of life
[35] integer, intent(in) :: board (:,:)
[36] integer            :: newboard (size(board, 1), size(board, 2))
[37] integer            :: i, j, neighbors ! loops
[38]
[39] newboard = 0 ! initialize next generation
[40] do i = 2, boardsize - 1
[41]   do j = 2, boardsize - 1
[42]     neighbors = sum (board (i - 1:i + 1, j - 1:j + 1)) %
[43]       - board (i, j)
[44]     if ( board (i, j) == 1 ) then ! life in the cell
```

```

[ 45]         if ( (neighbors > 3 .or. neighbors < 2) ) then
[ 46]             newboard (i, j) = 0           ! it died
[ 47]         else
[ 48]             newboard (i, j) = 1           ! newborn
[ 49]         end if ! on number of neighbors
[ 50]     else ! no life in the cell
[ 51]         if ( neighbors == 3 ) then
[ 52]             newboard (i, j) = 1           ! newborn
[ 53]         else
[ 54]             newboard (i, j) = 0           ! died
[ 55]         end if ! on number of neighbors
[ 56]     end if ! life status
[ 57] end do ! on column j
[ 58] end do ! on row i
[ 59] end function next_generation
[ 60]
[ 61] Subroutine spy (board) ! model matlab spy function
[ 62] ! Show an X at each non-zero entry of board, else show -
[ 63] integer, intent(in) :: board (:, :)
[ 64] character (len=1)   :: line (size(board, 1)) ! a line on screen
[ 65] integer             :: i                     ! loops
[ 66]
[ 67]     line = ' ' ! blank out the line
[ 68]     do i = 1, size (board, 1 ) ! loop over each row
[ 69]         line (1:size (board, 2 )) = '--' ! current board width
[ 70]         where ( board (i, :) /= 0 ) line = 'X' ! mark non-zero columns
[ 71]         write (*, '(80a1)') line ! print current row
[ 72]     end do ! over all rows
[ 73] end subroutine spy
[ 74] end program ! game_of_life
[ 75]
[ 76] ! Running gives:
[ 77] ! Initial Life Display:
[ 78] ! -----
[ 79] ! -----
[ 80] ! --X-----
[ 81] ! ---X-----
[ 82] ! -XXX-----
[ 83] ! -----
[ 84] ! -----
[ 85] ! -----
[ 86] ! -----
[ 87] ! -----
[ 88] ! Initially alive = 5
[ 89] !
[ 90] ! Enter number of generations to display: 4
[ 91] ! -----
[ 92] ! -----
[ 93] ! -----
[ 94] ! -X-X-----
[ 95] ! --XX-----
[ 96] ! --X-----
[ 97] ! -----
[ 98] ! -----
[ 99] ! -----
[100] ! -----
[101] !
[102] ! Generation number = 1
[103] ! Currently alive = 5
[104] ! continue? (y, n) n

```

C.4 Problem 2.5.1 : Conversion factors

This code illustrates the type of global units conversion factors that you can define for your field of study. They can be accessed by any program that includes a use `Conversion_Constants` line and cites a parameter name, as shown on line 16.

```

[ 1] Module Conversion_Constants ! DefineUnits Conversion
[ 2] ! Define selected precision
[ 3] INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 4] ! ===== Metric Conversions =====
[ 5] real(DP), parameter:: cm_Per_Inch = 2.54_DP
[ 6] real(DP), parameter:: kg_Per_Pound = 0.45359237_DP
[ 7] real(DP), parameter:: kg_Per_Short_Ton = 907.18474_DP
[ 8] real(DP), parameter:: kg_Per_Long_Ton = 1016.0469088_DP
[ 9] real(DP), parameter:: m_Per_Foot = 3.048_DP
[10] real(DP), parameter:: m_Per_Mile = 1609.344_DP
[11] real(DP), parameter:: m_Per_Naut_Mile = 1852.0_DP
[12] real(DP), parameter:: m_Per_Yard = 0.9144_DP
[13] end Module Conversion_Constants
[14] Program Test

```

```

[15] use Conversion_Constants
[16] print *, 'cm_Per_Inch = ', cm_Per_Inch ; End Program Test
[17] ! Running gives: cm_Per_Inch = 2.540000000000000004

```

This code illustrates the type of common physical constants that can be made available as global variables that you can define for your field of study. They can be accessed by any program that includes a use `Physical_Constants` line and cites a parameter name, as shown on line 60 below.

```

[ 1] Module Physical_Constants      ! Define Physical Constants
[ 2] ! Define selected precision
[ 3]     INTEGER, PARAMETER :: DP = KIND (1.d0) ! Alternate form
[ 4]
[ 5] ! ===== Physics Constants and units =====
[ 6] real(DP), parameter:: AMU_Value      = 1.6605402E-27_DP    ! kg
[ 7] real(DP), parameter:: Atmosphere_Pres = 9.80665E+04_DP    ! Pa
[ 8] real(DP), parameter:: Avogadro       = 6.0221367E+23_DP    ! 1/mol
[ 9] real(DP), parameter:: Bohr_Magneton  = 9.2740154E-24_DP    ! J/T
[10] real(DP), parameter:: Bohr_Radius    = 5.29177249E-11_DP    ! m
[11] real(DP), parameter:: Boltzmann      = 1.380657E-23_DP      ! J/K
[12] real(DP), parameter:: c_Light        = 2.997924580E+8_DP    ! m/s
[13] real(DP), parameter:: Electron_Compton = 2.42631058E-12_DP    ! m
[14] real(DP), parameter:: Electron_Angular = 5.2729E-35_DP      ! J*s
[15] real(DP), parameter:: Electron_Charge = -1.60217738E-19_DP    ! coul
[16] real(DP), parameter:: Electron_Mass_Rest = 9.1093897E-31_DP    ! kg
[17] real(DP), parameter:: Electron_Moment = 9.2847700E-24_DP    ! J/T
[18] real(DP), parameter:: Electron_Radius = 2.81794092E-15_DP    ! m
[19] real(DP), parameter:: Faraday        = 9.6485309E+04_DP    ! C/mo
[20] real(DP), parameter:: G_Universal    = 6.67260E-11_DP      ! m^3/(s^2*kg)
[21] real(DP), parameter:: Light_Year     = 9.46073E+15_DP      ! m
[22] real(DP), parameter:: Mech_equiv_Heat = 4.185E+3_DP        ! J/kcal
[23] real(DP), parameter:: Molar_Volume   = 0.02241410_DP      ! m^3/mol
[24] real(DP), parameter:: Neutron_Mass   = 1.6749286E-27_DP    ! kg
[25] real(DP), parameter:: Permeability   = 1.25663706143E-06_DP ! H/m
[26] real(DP), parameter:: Permittivity   = 8.85418781762E-12_DP ! F/m
[27] real(DP), parameter:: Planck_Const   = 6.6260754E-34_DP    ! J*s
[28] real(DP), parameter:: Proton_Mass    = 1.6726230E-27_DP    ! kg
[29] real(DP), parameter:: Proton_Moment  = 1.41060761E-26_DP    ! J/T
[30] real(DP), parameter:: Quantum_charge_r = 4.13556E+12_DP     ! J*s/C
[31] real(DP), parameter:: Rydberg_inf     = 1.0973731534E+07_DP ! 1/m
[32] real(DP), parameter:: Rydberg_Hydrogen = 1.09678E+07_DP     ! 1/m
[33] real(DP), parameter:: Std_Atmosphere = 1.01325E+05_DP     ! Pa
[34] real(DP), parameter:: Stefan_Boltzmann = 5.67050E-08_DP     ! W/(m^2*K^4)
[35] real(DP), parameter:: Thomson_cross_sect = 6.6516E-29_DP     ! m^2
[36] real(DP), parameter:: Universal_Gas_C = 8.314510_DP       ! J/mol*K
[37]
[38] ! ===== Astronomy Constants and units =====
[39] real(DP), parameter:: AU_Earth_Sun    = 1.4959787E+11_DP    ! m
[40] real(DP), parameter:: Anomal_Month   = 27.5546_DP        ! days
[41] real(DP), parameter:: Anomal_Year    = 365.2596_DP        ! days
[42] real(DP), parameter:: Dracon_Month   = 27.2122_DP        ! days
[43] real(DP), parameter:: Earth_G         = 9.80665_DP        ! m/s^2
[44] real(DP), parameter:: Earth_Mass     = 5.974E+24_DP      ! kg
[45] real(DP), parameter:: Earth_Radius_Eq = 6.37814E+6_DP     ! m
[46] real(DP), parameter:: Earth_Radius_Mean = 6.371E+6_DP     ! m
[47] real(DP), parameter:: Earth_Radius_Polar = 6.356755E+6_DP ! m
[48] real(DP), parameter:: Julian_Year     = 365.25_DP        ! days
[49] real(DP), parameter:: Rotation_Day    = 23.93447222_DP    ! hours
[50] real(DP), parameter:: Sidereal_Day    = 23.93446944_DP    ! hours
[51] real(DP), parameter:: Sidereal_Month   = 27.3217_DP        ! days
[52] real(DP), parameter:: Sidereal_Ratio  = 1.0027379092558_DP !
[53] real(DP), parameter:: Sidereal_Year   = 365.2564_DP        ! days
[54] real(DP), parameter:: Solar_Day       = 24.06571111_DP    ! hours
[55] real(DP), parameter:: Synodic_Month   = 29.5306_DP        ! days
[56] real(DP), parameter:: Tropical_Year   = 365.2422_DP        ! days
[57] end Module Physical_Constants      ! Define Physical Constants
[58] Program Test
[59]     use Physical_Constants
[60]     print *, 'Avogadro = ', Avogadro ; End Program Test
[61] ! Running gives: Avogadro = 0.6022136699999999967E+24

```

C.5 Problem 3.5.3 : Creating a vector class

We begin by defining the components to be included in our vector object. They include the length of each vector and a corresponding real array of pointers to the vector components:

```

[ 1] module class_Vector      ! filename: class_Vector.f90
[ 2] ! public, everything by default, but can specify any
[ 3]     implicit none
[ 4]     type Vector
[ 5]     private

```

```

[ 6]     integer                :: size    ! vector length
[ 7]     real, pointer, dimension(:) :: data ! component values
[ 8] end type Vector

```

For persons familiar with vectors the use of overloaded operators makes sense (but it often does not make sense). Thus we overload the addition, subtraction, multiplication, assignment, and logical equal to operators by defining the correct class members to be used for different argument types:

```

[ 9] ! Overload common operators
[10] interface operator (+) ! add others later
[11] module procedure add_Vector, add_Real_to_Vector; end interface
[12] interface operator (-) ! add unary versions later
[13] module procedure subtract_Vector, subtract_Real; end interface
[14] interface operator (*) ! overload *
[15] module procedure dot_Vector, real_mult_Vector, Vector_mult_real
[16] end interface
[17] interface assignment (=) ! overload =
[18] module procedure equal_Real; end interface
[19] interface operator (==) ! overload ==
[20] module procedure is_equal_to; end interface
[21]

```

Then we encapsulate the supporting member functions, beginning with two constructors, assign and make_Vector:

```

[22] contains ! functions & operators
[23]
[24] function assign (values) result (name) ! array to vector constructor
[25]   real, intent(in) :: values(:) ! given rank 1 array
[26]   integer          :: length    ! array size
[27]   type (Vector)    :: name      ! Vector to create
[28]   length = size(values); allocate ( name%data(length) )
[29]   name % size = length; name % data = values; end function assign
[30]
[31] function make_Vector (len, values) result(v) ! Optional Constructor
[32]   integer, optional, intent(in) :: len ! number of values
[33]   real, optional, intent(in) :: values(:) ! given values
[34]   type (Vector) :: v
[35]   if ( present (len) ) then ! create vector data
[36]     v%size = len ; allocate ( v%data(len) )
[37]     if ( present (values) ) then ; v%data = values ! vector
[38]     else ; v%data = 0.d0 ! null vector
[39]   end if ! values present
[40]   else ! scalar constant
[41]     v%size = 1 ; allocate ( v%data(1) ) ! default
[42]     if ( present (values) ) then ; v%data(1) = values(1) ! scalar
[43]     else ; v%data(1) = 0.d0 ! null
[44]   end if ! value present
[45]   end if ! len present
[46] end function make_Vector
[47]

```

The remainder of the members are given in alphabetical order:

```

[48] function add_Real_to_Vector (v, r) result (new) ! overload +
[49]   type (Vector), intent(in) :: v
[50]   real, intent(in) :: r
[51]   type (Vector) :: new ! new = v + r
[52]   if ( v%size < 1 ) stop "No sizes in add_Real_to_Vector"
[53]   allocate ( new%data(v%size) ) ; new%size = v%size
[54]   ! new%data = v%data + r ! as array operation
[55]   new%data(1:v%size) = v%data(1:v%size) + r ; end function
[56]
[57] function add_Vector (a, b) result (new) ! vector + vector
[58]   type (Vector), intent(in) :: a, b
[59]   type (Vector) :: new ! new = a + b
[60]   if ( a%size /= b%size ) stop "Sizes differ in add_Vector"
[61]   allocate ( new%data(a%size) ) ; new%size = a%size
[62]   new%data = a%data + b%data ; end function add_Vector

```

Note that lines 55 and 62 above are similar ways to avoid writing serial loops that would have to be used in most languages. This keeps the code cleaner and shorter, and more importantly it lets the compiler carry out those operations in parallel on some machines.

While copy members are very important to C++ programmers the following `copy_Vector` should probably be omitted since you would not usually pass big arrays as copies and F90 defaults to passing by reference unless forced to pass by value.

```

[63]
[64] function copy_Vector (name) result (new)
[65]   type (Vector), intent(in) :: name

```

```

[ 66] type (Vector)           :: new
[ 67]   allocate ( new%data(name%size) ) ; new%size = name%size
[ 68]   new%data = name%data      ; end function copy_Vector

```

The routine `delete_Vector` is the destructor for this class. In some sense it is incomplete because it does not delete the `size` attribute. It was decided that while the actual array of data may take a huge amount of storage, the single integer was not important. To be more complete one would have to have to make `size` an integer pointer and allocate and deallocate it at numerous locations within this module.

```

[ 69]
[ 70] subroutine delete_Vector (name) ! deallocate allocated items
[ 71]   type (Vector), intent(inout) :: name
[ 72]   integer                       :: ok ! check deallocate status
[ 73]   deallocate (name%data, stat = ok )
[ 74]   if ( ok /= 0 ) stop "Vector not allocated in delete_Vector"
[ 75]   name%size = 0 ; end subroutine delete_Vector
[ 76]
[ 77] function dot_Vector (a, b) result (c)      ! overload *
[ 78]   type (Vector), intent(in) :: a, b
[ 79]   real                       :: c
[ 80]   if ( a%size /= b%size ) stop "Sizes differ in dot_Vector"
[ 81]   c = dot_product(a%data, b%data); end function dot_Vector
[ 82]
[ 83] subroutine equal_Real (new, R) ! overload =, real to vector
[ 84]   type (Vector), intent(inout) :: new
[ 85]   real, intent(in) :: R
[ 86]   if ( associated (new%data) ) deallocate (new%data)
[ 87]   allocate ( new%data(1) ); new%size = 1
[ 88]   new%data = R ; end subroutine equal_Real
[ 89]
[ 90] logical function is_equal_to (a, b) result (t_f) ! overload ==
[ 91]   type (Vector), intent(in) :: a, b ! left & right of ==
[ 92]   t_f = .false. ! initialize
[ 93]   if ( a%size /= b%size ) return ! same size ?
[ 94]   t_f = all ( a%data == b%data ) ! and all values match
[ 95] end function is_equal_to
[ 96]
[ 97] function length (name) result (n) ! accessor member
[ 98]   type (Vector), intent(in) :: name
[ 99]   integer :: n
[100]   n = name % size ; end function length
[101]
[102] subroutine list (name) ! accessor member, for prettier printing
[103]   type (Vector), intent(in) :: name
[104]   print *, "[", name % data(1:name%size), "]" ; end subroutine list
[105]
[106] function normalize_Vector (name) result (new)
[107]   type (Vector), intent(in) :: name
[108]   type (Vector) :: new
[109]   real :: total, nil = epsilon(1.0) ! tolerance
[110]   allocate ( new%data(name%size) ); new%size = name%size
[111]   total = sqrt ( sum ( name%data**2 ) ) ! intrinsic functions
[112]   if ( total < nil ) then ; new%data = 0.d0 ! avoid division by 0
[113]   else ; new%data = name%data/total
[114]   end if ; end function normalize_Vector
[115]
[116] subroutine read_Vector (name) ! read array, assign
[117]   type (Vector), intent(inout) :: name
[118]   integer, parameter :: max = 999
[119]   integer :: length
[120]   read (*, '(il)', advance = 'no') length
[121]   if ( length <= 0 ) stop "Invalid length in read_Vector"
[122]   if ( length >= max ) stop "Maximum length in read_Vector"
[123]   allocate ( name % data(length) ) ; name % size = length
[124]   read *, name % data(1:length) ; end subroutine read_Vector
[125]
[126] function real_mult_Vector (r, v) result (new) ! overload *
[127]   real, intent(in) :: r
[128]   type (Vector), intent(in) :: v
[129]   type (Vector) :: new ! new = r * v
[130]   if ( v%size < 1 ) stop "Zero size in real_mult_Vector"
[131]   allocate ( new%data(v%size) ) ; new%size = v%size
[132]   new%data = r * v%data ; end function real_mult_Vector
[133]
[134] function size_Vector (name) result (n) ! accessor member
[135]   type (Vector), intent(in) :: name
[136]   integer :: n
[137]   n = name % size ; end function size_Vector
[138]
[139] function subtract_Real(v, r) result(new) ! vector-real, overload -
[140]   type (Vector), intent(in) :: v
[141]   real, intent(in) :: r
[142]   type (Vector) :: new ! new = v + r

```

```

[143]     if ( v%size < 1 ) stop "Zero length in subtract_Real"
[144]     allocate ( new%data(v%size) ) ; new%size = v%size
[145]     new%data = v%data - r           ; end function subtract_Real
[146]
[147] function subtract_Vector (a, b) result (new) ! overload -
[148] type (Vector), intent(in) :: a, b
[149] type (Vector)              :: new
[150] if ( a%size /= b%size ) stop "Sizes differ in subtract_Vector"
[151] allocate ( new%data(a%size) ) ; new%size = a%size
[152] new%data = a%data - b%data      ; end function subtract_Vector
[153]
[154] function values (name) result (array)      ! accessor member
[155] type (Vector), intent(in) :: name
[156] real                      :: array(name%size)
[157] array = name % data ; end function values

```

The routine `delete_Vector` is the manual constructor for this class. It has no optional arguments so both arguments must be supplied, and it duplicates the constructor on line 31, but it uses the naming convention preferred by the author.

```

[158]
[159] function Vector_ (length, values) result(name) ! constructor
[160] integer, intent(in) :: length      ! array size
[161] real, target, intent(in) :: values(length) ! given array
[162] real, pointer          :: pt_to_val(:) ! pointer to array
[163] type (Vector)         :: name       ! Vector to create
[164] integer               :: get_m     ! allocate flag
[165] allocate ( pt_to_val (length), stat = get_m ) ! allocate
[166] if ( get_m /= 0 ) stop 'allocate error'      ! check
[167] pt_to_val = values                          ! dereference values
[168] name = Vector(length, pt_to_val) ! intrinsic constructor
[169] end function Vector_
[170]
[171] function Vector_max_value (a) result (v)      ! accessor member
[172] type (Vector), intent(in) :: a
[173] real                      :: v
[174] v = maxval ( a%data(1:a%size) ) ; end function Vector_max_value
[175]
[176] function Vector_min_value (a) result (v)      ! accessor member
[177] type (Vector), intent(in) :: a
[178] real                      :: v
[179] v = minval ( a%data(1:a%size) ) ; end function Vector_min_value
[180]
[181] function Vector_mult_real(v, r) result(new) ! vec*real, overload *
[182] type (Vector), intent(in) :: v
[183] real, intent(in) :: r
[184] type (Vector)      :: new          ! new = v * r
[185] if ( v%size < 1 ) stop "Zero size in Vector_mult_real"
[186] new = Real_mult_Vector(r, v); end function Vector_mult_real
[187]
[188] end module class_Vector

```

A first test of this class is given below along with comments that give the verifications of the members.

```

[ 1] !           Testing Vector Class Constructors & Operators
[ 2] include 'class_Vector.f90'           ! see previous figure
[ 3] program check_vector_class
[ 4]   use class_Vector
[ 5]   implicit none
[ 6]
[ 7]   type (Vector) :: x, y, z
[ 8]
[ 9] !           test optional constructors: assign, and copy
[10] x = make_Vector ()           ! single scalar zero
[11] write (*, '("made scalar x = ")', advance='no'); call list(x)
[12]
[13] call delete_Vector (x) ; y = make_Vector (4) ! 4 zeros
[14] write (*, '("made null y = ")', advance='no'); call list(y)
[15]
[16] z = make_Vector (4, (/11., 12., 13., 14./) ) ! 4 non-zeros
[17] write (*, '("made full z = ")', advance='no'); call list(z)
[18] write (*, '("assign [ 31., 32., 33., 34. ] to x")')
[19]
[20] x = assign( (/31., 32., 33., 34./) )           ! (4) non-zeros
[21] write (*, '("assigned x = ")', advance='no'); call list(x)
[22]
[23] x = Vector_(4, (/31., 32., 33., 34./) )       ! 4 non-zeros
[24] write (*, '("public x = ")', advance='no'); call list(x)
[25] write (*, '("copy x to y = ")', advance='no')
[26] y = copy_Vector (x) ; call list(y)             ! copy
[27]
[28] !           test overloaded operators
[29] write (*, '("z * x gives ")', advance='no'); print *, z*x ! dot
[30] write (*, '("z + x gives ")', advance='no'); call list(z+x) ! add

```

```

[31] y = 25.6                                ! real to vector
[32] write (*,'("y = 25.6 gives ")',advance='no'); call list(y)
[33] y = z                                    ! equality
[34] write (*,'("y = z gives y as ")', advance='no'); call list(y)
[35] write (*,'("logic y == x gives ")',advance='no'); print *, y==x
[36] write (*,'("logic y == z gives ")',advance='no'); print *, y==z
[37]
[38] !                                     test destructor, accessors
[39] call delete_Vector (y)                  ! destructor
[40] write (*,'("deleting y gives y = ")',advance='no'); call list(y)
[41] print *, "size of x is ", length (x)    ! accessor
[42] print *, "data in x are [", values (x), "]" ! accessor
[43] write (*,'("2. times x is ")',advance='no'); call list(2.0*x)
[44] write (*,'("x times 2. is ")',advance='no'); call list(x*2.0)
[45] call delete_Vector (x); call delete_Vector (z) ! clean up
[46] end program check_vector_class
[47] ! Running gives the output:             ! made scalar x = [0]
[48] ! made null y = [0, 0, 0, 0]             ! made full z = [11, 12, 13, 14]
[49] ! assign [31, 32, 33, 34] to x           ! assigned x = [31, 32, 33, 34]
[50] ! public x = [31, 32, 33, 34]           ! copy x to y = [31, 32, 33, 34]
[51] ! z * x gives 1630                       ! z + x gives [42, 44, 46, 48]
[52] ! y = 256 gives [2560000004]           ! y = z, y = [11, 12, 13, 14]
[53] ! logic y == x gives F                   ! logic y == z gives T
[54] ! deleting y gives y = []               ! size of x is 4
[55] ! data in x : [31, 32, 33, 34]         ! 2 times x is [62, 64, 66, 68]
[56] ! x times 2 is [62, 64, 66, 68]

```

Having tested the vector class we will now use it in some typical vector operations. We want a program that will work with arrays of vectors to read in the number of vectors. The array of vectors will use an automatic storage mode. That could be risky because if the system runs out of memory we get a fatal error message and the run aborts. If we made the alternate choice of allocatable arrays then we could check the allocation status and have a chance (but not a good chance) of closing down the code in some "friendly" manner. Once the code reads the number of vectors then for each one it reads the number of components and the the component values. After testing some simple vector math we compute a more complicated result know as the orthonormal basis for the given set of vectors:

```

[ 1] ! Test Vector Class Constructors, Operators and Basis
[ 2] include 'class_Vector.f'
[ 3]
[ 4] program check_basis ! demonstrate a typical Vector class
[ 5]   use class_Vector
[ 6]   implicit none
[ 7]
[ 8]   interface
[ 9]     subroutine testing_basis (N_V)
[10]       integer, intent(in) :: N_V
[11]     end subroutine testing_basis
[12]   end interface
[13]
[14]   print *, "Test automatic allocate, deallocate"
[15]   print *, " "; read *, N_V
[16]   print *, "The number of vectors to be read is: ", N_V
[17]   call testing_basis ( N_V) ! to use automatic arrays
[18] end program check_basis
[19]
[20] subroutine testing_basis (N_V)
[21] ! test vectors AND demo automatic allocation/deallocation
[22] use class_Vector
[23]
[24] integer, intent(in) :: N_V
[25] type (Vector)      :: Input(N_V) ! automatic array
[26] type (Vector)      :: Ortho(N_V) ! automatic array
[27] integer            :: j
[28] real               :: norm
[29]
[30] interface
[31]   subroutine orthonormal_basis (Input, Ortho, N_given)
[32]     use class_Vector
[33]     type (Vector), intent(in)  :: Input(N_given)
[34]     type (Vector), intent(out) :: Ortho(N_given)
[35]     integer, intent(in)        :: N_given
[36]   end subroutine orthonormal_basis
[37] end interface
[38]
[39] print *, " "; print *, "The given ", N_V, " vectors:"
[40] do j = 1, N_V
[41]   call read_Vector ( Input(j) )
[42]   call list        ( Input(j) )
[43] end do ! for j
[44]
[45] print *, " "

```

```

[ 46] print *, "The Orthogonal Basis of the original set is:"
[ 47]
[ 48] call orthonormal_basis (Input, Ortho, N_V)
[ 49] do j = 1, N_V          ! list new orthogonal basis
[ 50]     call list ( Ortho(j) )
[ 51] end do ! for j
[ 52]
[ 53]     ! use vector class features & operators
[ 54] print *, ' ' ; print *, "vector 1 + vector 2 = "
[ 55] call list (Input(1)+Input(2))
[ 56] print *, "vector 1 - vector 2 = "
[ 57] call list (Input(1)-Input(2))
[ 58] print *, "vector 1 dot vector 2 = ", Input(1)*Input(2)
[ 59] print *, "vector 1 * 3.5 = "
[ 60] call list (3.5*Input(1))
[ 61] norm = sqrt ( dot_Vector( Input(1), Input(1) ) )
[ 62] print *, "norm(vector 1) = ", norm
[ 63] print *, "normalized vector 1 = "
[ 64] call list (normalize_Vector(Input(1)))
[ 65] print *, "max(vector 1) = ", vector_max_value (Input(1))
[ 66] print *, "min(vector 1) = ", vector_min_value (Input(1))
[ 67] print *, "length of vector 1 = ", length ( Input(1) )
[ 68] end subroutine testing_basis
[ 69]
[ 70] subroutine orthonormal_basis (Input, Ortho, N_given)
[ 71] ! Find Orthonormal Basis of a Set of Vector Classes
[ 72] use class_Vector
[ 73] !*****
[ 74] ! =, -, +, * are overloaded operators from class_Vector
[ 75] !*****
[ 76]
[ 77] type (Vector), intent(in) :: Input(N_given)
[ 78] type (Vector), intent(out) :: Ortho(N_given)
[ 79] integer, intent(in) :: N_given
[ 80] integer :: i, j ! loops
[ 81] real :: dot
[ 82] do i = 1, N_given ! original set of vectors
[ 83]     Ortho(i) = Input(i) ! copy input vector class
[ 84]     do j = 1, i ! for previous copies
[ 85]         dot = dot_Vector(Ortho(i), Ortho(j))
[ 86]         Ortho(i) = Ortho(i) - (dot*Ortho(j))
[ 87]     end do ! for j
[ 88]     Ortho(i) = normalize_Vector ( Ortho(i) )
[ 89] end do ! over i
[ 90] end subroutine orthonormal_basis
[ 91]
[ 92] ! Compiling and inputting :
[ 93] ! 4
[ 94] ! 3 0.625 0 0
[ 95] ! 3 7.5 3.125 0
[ 96] ! 3 13.25 -7.8125 6.5
[ 97] ! 3 14.0 3.5 -7.5
[ 98] ! Gives:
[ 99] ! Test automatic allocate, deallocate
[100] !
[101] ! The number of vectors to be read is: 4
[102] ! The given 4 vectors:
[103] ! [ 0.6250 0.0000 0.0000 ]
[104] ! [ 7.5000 3.1250 0.0000 ]
[105] ! [ 13.2500 -7.8125 6.5000 ]
[106] ! [ 14.0000 3.5000 -7.5000 ]
[107] !
[108] ! The Orthogonal Basis of the original set is:
[109] ! [ 1.0000 0.0000 0.0000 ]
[110] ! [ 0.0000 -1.0000 0.0000 ]
[111] ! [ 0.0000 0.0000 -1.0000 ]
[112] ! [ 0.0000 0.0000 0.0000 ]
[113] !
[114] ! vector 1 + vector 2 = [ 8.1250 3.1250 0.0000 ]
[115] ! vector 1 - vector 2 = [-6.8750 -3.1250 0.0000 ]
[116] ! vector 1 dot vector 2 = 4.6875
[117] ! vector 1 * 3.5 = [ 2.1875 0.0000 0.0000 ]
[118] ! norm(vector 1) = 0.6250
[119] ! normalized vector 1 = [ 1.0000 0.0000 0.0000 ]
[120] ! max(vector 1) = 0.6250
[121] ! min(vector 1) = 0.0000
[122] ! length of vector 1 = 3

```

C.6 Problem 3.5.4 : Creating a sparse vector class

This class begins like the previous Vector class except that we must add a row entry (line 4) for each data value entry (line 5). This is done for efficiency since we expect most values in sparse vectors to be zero (and hence their name). The attribute `non_zero` is the size of both rows and values.

```
[ 1] module class_sparse_Vector
[ 2]   implicit none
[ 3]   type sv           ! a sparse vector
[ 4]   integer          :: non_zeros
[ 5]   integer, pointer :: rows(:)
[ 6]   real,    pointer :: values(:)
[ 7] end type
[ 8]
```

The overloading process is similar, but now we will see that much more logic is required to deal with the zero entries and new zeros created by addition or multiplication.

```
[ 8]   interface assignment (=)
[ 9]     module procedure equal_Vector ; end interface
[10]   interface operator (.dot.) ! define dot product operator
[11]     module procedure dot_Vector ; end interface
[12]   interface operator (==) ! Boolean equal to
[13]     module procedure is_equal_to ; end interface
[14]   interface operator (*) ! term by term product
[15]     module procedure el_by_el_Mult, real_mult_Sparse
[16]     module procedure Sparse_mult_real
[17]   end interface
[18]   interface operator (-) ! for sparse vectors
[19]     module procedure Sub_Sparse_Vectors ; end interface
[20]   interface operator (+) ! for sparse vectors
[21]     module procedure Sum_Sparse_Vectors ; end interface
[22]
[23] contains ! operators and functionality
```

In the following constructor for the class note that both the pointer array attributes are allocated (line 32) the same amount of storage in memory. One should also include the allocation status flag here and checks its value to raise a possible exception (as seen in lines 41-46).

```
[24]   subroutine make_Sparse_Vector (s,n,r,v)
[25]     ! allows zero length vectors
[26]     type (sv) :: s ! name
[27]     integer, intent(in) :: n ! size
[28]     integer, intent(in) :: r(n) ! rows
[29]     real,    intent(in) :: v(n) ! values
[30]     if ( n < 0 ) stop &
[31]         "Error, negative rows in make_Sparse_Vector"
[32]     allocate (s%rows(n), s%values(n))
[33]     s%non_zeros = n ! copy size
[34]     s%rows      = r ! row array assignment
[35]     s%values    = v ! value array assignment
[36]   end subroutine make_Sparse_Vector
[37]
```

This is really a destructor. Again, it is incomplete because the integer array size was not made allocatable for simplicity.

```
[38]   subroutine delete_Sparse_Vector (s)
[39]     type (sv) :: s ! name of sparse vector
[40]     integer :: error ! deallocate status flag, 0 no error
[41]     deallocate (s%rows, s%values, stat = error) ! memory released
[42]     if ( error == 0 ) then
[43]       s%non_zeros = 0 ! reset size
[44]     else ! never created
[45]       stop "Sparse vector to delete does not exist"
[46]     end if ; end subroutine delete_Sparse_Vector
[47]
```

This creates a user defined operator call `.dot.` to be applied to sparse vectors.

```
[48]   function dot_Vector (u, v) result (d) ! defines .dot.
[49]     ! dot product of sparse vectors
[50]     type (sv), intent(in) :: u, v ! sparse vectors
[51]     type (sv)           :: w ! sparse vector, temporary
[52]     real                :: d ! dot product value
[53]     d = 0.0 ! default
[54]     if ( u%non_zeros < 1 .or. v%non_zeros < 1 ) return ! null
[55]     w = el_by_el_Mult (u, v) ! element by element sparse product
[56]     if ( w%non_zeros > 0 ) &
[57]       d = sum( w%values(1:w%non_zeros) ) ! summed
[58]     call delete_Sparse_Vector (w) ! delete temp
```

```
[ 59]     end function dot_Vector
[ 60]
```

The above `dot_Vector` is more complicated in this format because it is likely that stored non-zero values will be multiplied by (unstored) zeros. Thus, the real work is done in the following member function that employs Boolean logic. The terms for the summation that creates the scalar dot product are first computed in a full vector equal in length to the minimum row number given. Observe that its size is established through the use of the `min` intrinsic, acting on the two given sizes, within the `dimension` attribute for the full array (lines 67,68). Three logical arrays (line 68) are used as “masks” which are `true` when a non-zero exists in the corresponding row of their associated sparse vector (down to the minimum row cited above). The three logical vectors are initialized in lines 77 to 92. That process ends with the third vector being created as a Boolean product (line 91) and the maximum possible number of non-zero products is found from the `count` intrinsic (line 92).

It is also important to note that the working space vector `full` is an `automatic` array and memory for it is automatically allocated for it each time the function is called. It could be an extremely long vector and thus it is possible (but not likely) that there would not be enough memory available. Then the system would abort with an error message. To avoid that possibility one could have declared `full` to be an `allocatable` vector and then allocate its memory by using a similar `min` construct. That allocation request should (always) include the `STAT` flag so that if the memory allocation fails it would be possible to issue an exception to try to avoid a fatal crash of the system (not likely).

```
[ 61]     function el_by_el_Mult ( u, v ) result ( w ) ! defines * operator
[ 62]     ! element by element product of sparse vectors: 0 * real ?
[ 63]     type ( sv ), intent ( in ) :: u, v           ! given vectors
[ 64]     type ( sv ) :: w                          ! new vector
[ 65]     real :: full ( min ( u%rows ( u%non_zeros ), & ! automatic
[ 66]     & v%rows ( v%non_zeros ) ) ) ! workspace
[ 67]     logical, dimension ( min ( u%rows ( u%non_zeros ), &
[ 68]     v%rows ( v%non_zeros ) ) ) :: u_m, v_m, w_m ! logical product masks
[ 69]     integer :: j, k, last, n, row
[ 70]     ! is either u or v null ?
[ 71]     if ( u%non_zeros < 1 .or. v%non_zeros < 1 ) then ! w is null
[ 72]         allocate ( w%rows ( 0 ), w%values ( 0 ) )
[ 73]         w%non_zeros = 0
[ 74]         return ! a null sparse vector
[ 75]     end if ! no calculation necessary
[ 76]
[ 77]     ! Initialize logic masks
[ 78]     last = min ( u%rows ( u%non_zeros ), v%rows ( v%non_zeros ) ) ! max size
[ 79]     u_m = .false. ! assume no contributions
[ 80]     do j = 1, size ( u%rows )
[ 81]         row = u%rows ( j ) ! get row number to flag
[ 82]         if ( row > last ) exit ! j loop
[ 83]         u_m ( row ) = .true. ! possible contribution
[ 84]     end do ! to initialize u mask
[ 85]     v_m = .false. ! assume no contributions
[ 86]     do j = 1, size ( v%rows )
[ 87]         row = v%rows ( j ) ! get row number to flag
[ 88]         if ( row > last ) exit ! j loop
[ 89]         v_m ( row ) = .true. ! possible contribution
[ 90]     end do ! to initialize v mask
[ 91]     w_m = ( u_m .and. v_m ) ! Boolean product logic
[ 92]     n = count ( w_m ) ! count possible products
[ 93]     ! if ( n == 0 ) print *, "Warning: zero length sparse" ! debug
[ 94]
```

The vector `full` is set to zero (line 96) and comparison DO loops (lines 97,101) over the two given vectors are minimized (lines 100,103) by testing where the mask vector `w_m` is `true` (thereby indicating a non-zero product). When all the products are stored in the `full` vector it is converted to the sparse vector storage mode (line 109) for release as the return result. Because `full` is an `automatic` array its memory is automatically released when the function is exited.

```
[ 95]     ! Fill the product workspace, full
[ 96]     full = 0.0 ! initialize
[ 97]     do j = 1, size ( u%rows ) ! loop over u
[ 98]         row = u%rows ( j ) ! row in u
[ 99]         if ( row > last ) exit ! this loop in u ! past end of w
[100]        if ( .not. w_m ( row ) ) cycle ! to next j ! not in product
[101]        do k = 1, size ( v%rows ) ! loop over v
[102]            if ( v%rows ( k ) > last ) exit ! this loop ! past end of w
[103]            if ( .not. w_m ( v%rows ( k ) ) ) cycle ! to k+1 ! not in product
[104]            if ( row == v%rows ( k ) ) then ! same row, u & v
[105]                full ( row ) = u%values ( j ) * v%values ( k ) ! get product
```

```

[106]         end if
[107]     end do ! on k in v
[108] end do ! on j in u
[109] w = Vector_To_Sparse (full) !delete any zeros
[110] end function el_by_el_Mult ! deletes full & 3 masks
[111]

```

The operator overloading members are given with the next function (line 112) as well as in lines 140, 231, and 320.

```

[112] subroutine equal_Vector (new, s) ! overload =
[113] type (sv), intent(inout) :: new
[114] type (sv), intent(in)   :: s
[115] allocate ( new%rows(s%non_zeros) )
[116] allocate ( new%values(s%non_zeros) )
[117] new%non_zeros = s%non_zeros
[118] if ( s%non_zeros > 0 ) then
[119]     new%rows(1:s%non_zeros) = s%rows(1:s%non_zeros) ! array copy
[120]     new%values(1:s%non_zeros) = s%values(1:s%non_zeros) ! copy
[121] end if ; end subroutine equal_Vector
[122]
[123] function get_element (name, row) result (v)
[124] type (sv), intent(in) :: name ! sparse vector
[125] integer, intent(in) :: row ! row in sparse vector
[126] integer :: j ! loops
[127] real :: v ! value at row
[128] v = 0.0 ! default
[129] if ( row < 1 ) stop "Invalid row number, get_element"
[130] if ( name%non_zeros < 1 ) return ! not here
[131] if ( row > name%rows(name%non_zeros) ) return ! not here
[132] do j = 1, name%non_zeros
[133]     if ( row == name%rows(j) ) then
[134]         v = name%values(j) ! found the value
[135]         return ! search done
[136]     end if ! in the vector
[137] end do ! over possible values
[138] end function get_element
[139]
[140] function is_equal_to (a, b) result (t_or_f) ! define ==
[141] type (sv), intent(in) :: a, b ! two sparse vectors
[142] logical :: t_or_f
[143] integer :: i ! loops
[144] t_or_f = .true. ! default
[145] if ( a%non_zeros == b%non_zeros ) then ! also check values
[146]     do i = 1, a%non_zeros ! or use count function for simplicity
[147]         if ( a%rows(i) /= b%rows(i) .or. &
[148]             a%values(i) /= b%values(i) ) then
[149]             t_or_f = .false. ! because rows and/or values differ
[150]             return ! no additional checks needed
[151]         end if ! same values
[152]     end do ! over sparse rows
[153] else ! sizes differ so vectors must be different
[154]     t_or_f = .false.
[155] end if ! sizes match
[156] end function is_equal_to
[157]
[158] function largest_index (s) result(row)
[159] type (sv), intent(in) :: s ! sparse vector
[160] integer :: row ! last non-zero in full vector
[161] integer :: j ! loops
[162] row = 0 ! initialize
[163] if ( s%non_zeros < 1 ) return ! null vector
[164] do j = s%non_zeros, 1, -1 ! loop backward
[165]     if ( s%values(j) /= 0.0 ) then ! last non-zero term
[166]         row = s%rows(j) ! actual row number
[167]         return ! search done
[168]     end if
[169] end do
[170] end function largest_index
[171]
[172] function length (name) result (n)
[173] type (sv), intent(in) :: name
[174] integer :: n
[175] n = name % non_zeros ! read access to size, if private
[176] end function length
[177]

```

Once again we observe that the next two functions employ the colon operator (lines 185,196,199,201) to avoid explicit serial loops which would make them faster on certain vector and parallel computers.

```

[178] function norm (name) result (total)
[179] type (sv), intent(in) :: name
[180] real :: total
[181] if ( name%non_zeros < 1 ) then

```

```

[182]         ! print *, "Warning: empty vector in norm"
[183]         total = 0.0
[184]     else
[185]         total = sqrt( sum( name%values(1:name%non_zeros)**2 ) )
[186]     end if ! a null vector
[187] end function norm
[188]
[189] function normalize_Vector (s) result (new)
[190] type (sv), intent(in) :: s
[191] type (sv)              :: new
[192] real                   :: total, epsilon = 1.e-6
[193] allocate ( new%rows (s%non_zeros) )
[194] allocate ( new%values(s%non_zeros) )
[195] new%non_zeros          = s%non_zeros          ! copy size
[196] new%rows(1:s%non_zeros) = s%rows(1:s%non_zeros) ! copy rows
[197] total = sqrt( sum( s%values(1:s%non_zeros)**2 ) ) ! norm
[198] if ( total <= epsilon ) then                    ! divide by 0 ?
[199]     new%values(1:s%non_zeros) = 0.d0           ! set to zero
[200] else                                           ! or real values
[201]     new%values(1:s%non_zeros) = s%values(1:s%non_zeros)/total
[202] end if ! division by zero
[203] end function normalize_Vector
[204]
[205] subroutine pretty (s) ! print all values if space allows
[206] type (sv), intent(in) :: s ! sparse vector
[207] integer, parameter    :: limit = 20 ! for print size
[208] integer               :: n
[209] real                  :: full( s%rows(s%non_zeros) ) ! temp
[210] n = s%non_zeros
[211] if ( s%non_zeros < 1 .or. s%rows(s%non_zeros) > limit ) then
[212]     print *, "Wrong size to pretty print"
[213] else
[214]     full = 0. ! initialize to zero
[215]     if ( n > 0 ) full(s%rows) = s%values ! array copy non zeros
[216]     print *, "[", full, "]" ! pretty print
[217] end if ; end subroutine pretty ! automatic deallocate of full
[218]
[219] subroutine read_Vector (name) ! sparse vector data on unit 1
[220] type (sv), intent(inout) :: name
[221] integer :: length, j
[222] read (1, '(i1)', advance = 'no') length
[223] if ( length <= 0 ) stop "Invalid length in read_Vector"
[224] name % non_zeros = length
[225] allocate ( name % rows (length) )
[226] allocate ( name % values (length) )
[227] read (1,*) ( name%rows(j), name%values(j), j = 1, length)
[228] name%rows = name%rows + 1 ! default to 1 not 0 in F90
[229] end subroutine read_Vector
[230]
[231] function real_mult_Sparse (a, b) result (new)
[232] ! scalar * vector
[233] real, intent(in) :: a
[234] type (sv), intent(in) :: b
[235] type (sv) :: new
[236] allocate ( new%rows (b%non_zeros) )
[237] allocate ( new%values(b%non_zeros) )
[238] new%non_zeros = b%non_zeros
[239] if ( b%non_zeros < 1 ) then
[240]     print *, "Warning: zero size in real_mult_Sparse "
[241] else ! copy array components
[242]     new%rows (1:b%non_zeros) = b%rows (1:b%non_zeros)
[243]     new%values(1:b%non_zeros) = a * b%values(1:b%non_zeros)
[244] end if ! null vector
[245] end function real_mult_Sparse
[246]
[247] function rows_of (s) result(n) ! copy rows array of s
[248] type (sv) :: s ! sparse vector
[249] integer :: n(s%non_zeros) ! standard array
[250] if ( s%non_zeros < 1 ) stop "No rows to extract, rows_of"
[251] n = s%rows ! array copy
[252] end function rows_of
[253]
[254] subroutine set_element (s, row, value)
[255] ! Set, or insert, value into row of a sparse vector, s
[256] type (sv), intent(inout) :: s ! sparse vector
[257] integer, intent(in) :: row ! full vector row
[258] real, intent(in) :: value ! full vector value
[259] type (sv) :: new ! workspace
[260] logical :: found ! true if row exists
[261] integer :: j, where ! loops, locator
[262] found = .false. ! initialize
[263] where = 0 ! initialize
[264] do j = 1, s%non_zeros
[265]     if ( s%rows(j) == row ) then ! found it
[266]         s%values(j) = value ! value changed

```

```

[267]         return                                     ! no insert needed
[268]     end if
[269]     if ( s%rows(j) > row ) then
[270]         where = j                                     ! insert before j
[271]         exit ! the loop search
[272]     else ! s%rows(j) < row,                             may be next or last
[273]         where = j + 1
[274]     end if
[275] end do ! over current rows in s
[276] if ( .not. found ) then ! expand and insert at where
[277]     if ( where == 0 ) stop "Logic error, set_element"
[278]     new%non_zeros = s%non_zeros + 1
[279]     allocate ( new%rows (new%non_zeros) )
[280]     allocate ( new%values(new%non_zeros) )
[281]     ! copy preceeding rows
[282]     if ( where > 1 ) then ! copy to front of new
[283]         new%rows (1:where-1) = s%rows (1:where-1) ! array copy
[284]         new%values(1:where-1) = s%values(1:where-1) ! array copy
[285]     end if ! copy to front of new
[286]     ! insert, copy following rows of s
[287]     new%rows (where ) = row                               ! insert
[288]     new%values(where ) = value                           ! insert
[289]     new%rows (where+1:) = s%rows (where:)                ! array copy
[290]     new%values(where+1:) = s%values(where:)              ! array copy
[291]     ! deallocate s, move new to s, deallocate new
[292]     call delete_Sparse_Vector (s)                       ! delete s
[293]     call equal_Vector (s, new)                           ! s <- new
[294]     call delete_Sparse_Vector (new)                      ! delete new
[295] end if ! an insert is required
[296] end subroutine set_element
[297]
[298] subroutine show (s) ! alternating row number and value
[299]     type (sv) :: s ! sparse vector
[300]     integer :: j, k ! implied loops
[301]     k = length (s)
[302]     if ( k == 0 ) then
[303]         print *, k ; else ;
[304]         print *, k, ( s%rows(j)-1), s%values(j), j = 1, k )
[305]     end if ; end subroutine show
[306]
[307] subroutine show_r_v (s) ! all rows then all values
[308]     type (sv) :: s ! sparse vector
[309]     print *, "Vector has ", s%non_zeros, " non_zero terms."
[310]     if ( s%non_zeros > 0 ) then
[311]         print *, "Rows: ", s%rows - 1 ! to look like C++
[312]         print *, "Values: ", s%values
[313]     end if ; end subroutine show_r_v
[314]
[315] function size_of (s) result(n)
[316]     type (sv) :: s
[317]     integer :: n
[318]     n = s%non_zeros ; end function size_of
[319]
[320] function Sparse_mult_real (a, b) result (new)
[321]     ! vector * scalar
[322]     real, intent(in) :: b
[323]     type (sv), intent(in) :: a
[324]     type (sv) :: new
[325]     new = real_mult_Sparse ( b, a ) ! reverse the order
[326] end function Sparse_mult_real
[327]

```

In the following subtraction and addition functions we again note that sparse terms with the same values but opposite signs can result in new zero terms in the resulting vector. A temporary automatic workspace vector, `full`, is used to hold the preliminary results. In this case it must have a size that is the maximum of the two given vectors. Thus, the `max` intrinsic is employed in its dimension attribute (lines 331,344) which is opposite the earlier multiplication example (line 65).

```

[328] function Sub_Sparse_Vectors (u, v) result (w) ! defines -
[329]     type (sv), intent(in) :: u, v
[330]     type (sv) :: w
[331]     real :: full( max( u%rows(u%non_zeros), & ! automatic
[332]         & v%rows(v%non_zeros) ) ) ! workspace
[333]     if ( u%non_zeros <= 0 ) stop "First vector doesn't exist"
[334]     if ( v%non_zeros <= 0 ) stop "Second vector doesn't exist"
[335]     full = 0.0 ! set to zero
[336]     full(u%rows) = u%values ! copy first values
[337]     full(v%rows) = full(v%rows) - v%values ! less second values
[338]     w = Vector_To_Sparse (full) ! delete any zeros
[339] end function Sub_Sparse_Vectors ! automatically deletes full
[340]
[341] function Sum_Sparse_Vectors (u, v) result (w) ! defines +
[342]     type (sv), intent(in) :: u, v

```

```

[343]     type (sv)                :: w
[344]     real :: full( max( u%rows(u%non_zeros), & ! automatic
[345]     & v%rows(v%non_zeros) ) ) ! workspace
[346]     if ( u%non_zeros <= 0 ) stop "First vector doesn't exist"
[347]     if ( v%non_zeros <= 0 ) stop "Second vector doesn't exist"
[348]     full = 0.                ! set to zero
[349]     full(u%rows) = u%values   ! copy first values
[350]     full(v%rows) = full(v%rows) + v%values ! add second values
[351]     w = Vector_To_Sparse (full) ! delete any zeros
[352] end function Sum_Sparse_Vectors ! automatically deletes full
[353]
[354] function values_of (s) result(v) ! copy values of s
[355] type (sv) :: s ! sparse vector
[356] real :: v(s%non_zeros) ! standard array
[357] if ( s%non_zeros < 1 ) &
[358] stop "No values to extract, in values_of"
[359] v = s%values ! array copy
[360] end function values_of
[361]
[362] function Vector_max_value (a) result (v)
[363] type (sv), intent(in) :: a
[364] real :: v
[365] v = maxval (a%values(1:a%non_zeros)) ! intrinsic function
[366] ! is it a sparse vector with a false negative maximum ?
[367] if ( a%non_zeros < a%rows(a%non_zeros) .and. v < 0. ) v = 0.0
[368] end function Vector_max_value
[369]
[370] function Vector_min_value (a) result (v)
[371] type (sv), intent(in) :: a
[372] real :: v
[373] v = minval ( a%values(1:a%non_zeros) ) ! intrinsic function
[374] ! is it a sparse vector with a false positive minimum ?
[375] if ( a%non_zeros < a%rows(a%non_zeros) &
[376] .and. v > 0. ) v = 0.0
[377] end function Vector_min_value
[378]

```

This function is invoked several times in other member functions. It simply accepts a standard (dense) vector and converts it to the sparse storage mode in the return result.

```

[379] function Vector_To_Sparse (full) result (sparse)
[380] real, intent(in) :: full(:) ! standard array
[381] type (sv) :: sparse ! sparse vector copy
[382] integer :: j, n, number ! loops and counters
[383] n = count ( full /= 0.0 ) ! count non_zeros
[384] ! if ( n == 0 ) print *, "Warning: null full vector "
[385] allocate ( sparse%rows(n), sparse%values(n) )
[386] sparse%non_zeros = n ! sparse size
[387] number = 0 ! non zeros inserted
[388] do j = 1, size(full)
[389] if ( full(j) == 0.0 ) cycle ! to next j value
[390] number = number + 1 ! non zeros inserted
[391] sparse%rows(number) = j ! row number in full
[392] sparse%values(number) = full(j) ! value
[393] if ( number == n ) exit ! all non_zeros found
[394] end do ; end function Vector_To_Sparse
[395]
[396] function zero_sparse () result (s)
[397] type (sv) :: s ! create sparse null vector
[398] s%non_zeros = 0
[399] allocate (s%rows(0), s%values(0)); end function zero_sparse
[400] end module class_sparse_Vector

```

C.7 Problem 4.11.1 : Count the lines in an external file

```

[ 1] function inputCount(unit) result(linesOfInput)
[ 2] !-----
[ 3] ! takes a file number, counts the number of lines in that
[ 4] ! file, and returns the number of lines.
[ 5] !-----
[ 6] implicit none
[ 7] integer, intent(in) :: unit ! file unit number
[ 8] integer :: linesOfInput ! result
[ 9] integer ioResult ! system I/O action error code
[10] character temp ! place to hold the character read
[11]
[12] rewind (unit) ! go to the front of the file
[13] linesOfInput = 0 ! initially, there are 0 lines
[14]
[15] do ! Until iostat says we've hit the end_of_file
[16] read (unit,'(A)', iostat = ioResult) temp ! one char

```

```

[17]
[18]     if ( ioResult == 0 ) then           ! there were no errors
[19]         linesOfInput = linesOfInput + 1 ! increment lines
[20]     else if ( ioResult < 0 ) then      ! we've hit end-of-file
[21]         exit                          ! so exit this loop.
[22]     else ! ioResult is positive, which is a user error
[23]         write (*,*) 'inputCount: no data at unit =', unit
[24]         stop 'user read error'
[25]     end if
[26] end do
[27] rewind(unit)                ! go to the front of the file
[28] end Function inputCount

```

C.8 Problem 4.11.3 : Computing CPU time usage

While this is mainly designed to show the use of the module `tic_toc` you should note that the intrinsic way of printing a date or time is not “pretty” and could be easily improved.

```

[ 1] program watch
[ 2] ! -----
[ 3] ! Exercise DATE_AND_TIME and SYSTEM_CLOCK functions.
[ 4] ! -----
[ 5] use tic_toc
[ 6] implicit none
[ 7] character* 8 :: the_date
[ 8] character*10 :: the_time
[ 9] integer      :: j, k
[10] !
[11]   call date_and_time ( DATE = the_date )
[12]   call date_and_time ( TIME = the_time )
[13]   print *, 'The date is ', the_date, &
[14]   & ' and the time is now ', the_time
[15] !   Display facts about the system clock.
[16]   print *, ' '
[17]   call system_clock ( COUNT_RATE = rate )
[18]   print *, 'System clock runs at ', rate,&
[19]   & ' ticks per second'
[20] !
[21] !   Call the system clock to start an execution timer.
[22]   call tic
[23] !
[24] !   call run_the_job, or test with next 3 lines
[25]   do k = 1, 9999
[26]     j = sqrt ( real(k*k) )
[27]   end do
[28] !   Stop the execution timer and report execution time.
[29]   print *, ' '
[30]   print *, 'Job took ', toc (), ' seconds to execute.'
[31] end program watch           ! Running gives
[32] ! The date is 19980313 and the time is now 171837.792
[33] ! System clock runs at 100 ticks per second
[34] ! Job took 0.9999999776E-02 seconds to execute.

```

C.9 Problem 4.11.4 : Converting a string to upper case

The change from the `to_lower` should be obvious here. It seems desirable to place these two routines, and others that deal with strings into a single strings utility module.

```

[ 1] function to_upper (string) result (new_string) ! like C
[ 2] ! -----
[ 3] !           Convert a string or character to upper case
[ 4] !           (valid for ASCII or EBCDIC processors)
[ 5] ! -----
[ 6] implicit none
[ 7] character (len = *) , intent(in) :: string      ! unknown length
[ 8] character (len = len(string))   :: new_string ! same length
[ 9] character (len = 26), parameter ::           &
[10]     UPPER = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', &
[11]     lower = 'abcdefghijklmnopqrstuvwxyz'
[12] integer :: k ! loop counter
[13] integer :: loc ! position in alphabet
[14] new_string = string ! copy everything
[15] do k = 1, len(string) ! to change letters
[16]   loc = index ( lower, string(k:k)) ! locate
[17]   if (loc /= 0) new_string(k:k) = UPPER(loc:loc) ! convert
[18] end do ! over string characters
[19] end function to_upper

```

C.10 Problem 4.11.8 : Read two values from each line of an external file

```
[ 1] subroutine readData (inFile, lines, x, y)
[ 2] ! -----
[ 3] !   Take a file number, the number of lines to be read,
[ 4] !   and put the data into the arrays x and y
[ 5] ! -----
[ 6] ! inFile      is unit number to be read
[ 7] ! lines       is number of lines in the file
[ 8] ! x           is independent data
[ 9] ! y           is dependent data
[10] implicit none
[11] integer, intent(in)  :: inFile, lines
[12] real,   intent(out) :: x(lines), y(lines)
[13] integer                :: j
[14]
[15] rewind (inFile)           ! go to front of the file
[16] do j = 1, lines          ! for the entire file
[17]   read (inFile, *) x(j), y(j) ! get the x and y values
[18] end do ! over all lines
[19] end subroutine readData
```

C.11 Problem 4.11.14 : Two line least square fits

The extension of the single-line least squares fit shown in Fig. 4.21 is rather straightforward in that we will call subroutine `lsq_fit` multiple times. In line 37 we first call it in case a single-line fit may be more accurate than the expected two-line fit.

```
[ 1] program two_line_lsq_fit
[ 2] ! -----
[ 3] ! Best two-line linear least-squares fit of data in
[ 4] ! file specified by the user, and split in two sets
[ 5] ! -----
[ 6] implicit none
[ 7] real, allocatable :: x(:) ! independent data
[ 8] real, allocatable :: y(:) ! dependent data
[ 9]
[10] real :: fit(3), fit1(3), fit2(3) ! error results
[11] real :: left(3), right(3)       ! best results
[12] real :: error                  ! current error
[13] real :: error_min              ! best error
[14] integer :: split               ! best division
[15]
[16] integer, parameter :: filenumber = 1 ! input unit
[17] character (len = 64) :: filename    ! input file
[18] integer                :: lines     ! of input
[19] integer                :: inputCount, j ! loops
[20]
[21] ! Get the name of the file containing the data.
[22] write (*, *) 'Enter the data input filename:'
[23] read (*, *) filename
[24]
[25] ! Open that file for reading.
[26] open (unit = filenumber, file = filename)
[27]
[28] ! Find the number of lines in the file
[29] lines = inputCount (filenumber)
[30] write (*, *) 'There were ', lines, ' records read.'
[31]
[32] ! Allocate that many entries in the x and y array
[33] allocate (x(lines), y(lines))
[34] call read_xy_file (filenumber, lines, x, y) ! Read data
[35] close (filenumber)
[36]
[37] call lsq_fit (lines, x, y, fit) ! single line fit
[38] print *, "Single line fit"
[39] print *, "the slope is ", fit(1)
[40] print *, "the intercept is ", fit(2)
[41] print *, "the error is ", fit(3)
[42]
```

After that we want to try all the reasonable choices for breaching the data set into two adjacent regions that are each to be fit with a different straight line. Trial variables were defined in lines 10 and 12, while the best results found are in variables declared in lines 11, 13, and 14. Note that on line 48 we have required that at least three points be used to define an approximate straight line. If we allowed two points to be employed we would get a false (or misleading) indication of zero error for such a choice. Thus, in

line 48 we begin a loop over all possible sets of three or more data points and call `lsq_fit` for each of the two segments, as seen in lines 50 and 51.

```
[ 43] ! Loop to determine the mean squared error for each
[ 44] ! of the possible two divisions of the data
[ 45] !
[ 46]   error_min = HUGE(error_min)    ! initialize the error_min
[ 47]   split = 3                      ! initialize split point
[ 48]   do j = 3, lines-3             ! 3 pts to approximate a line
[ 49]   !   least-squares fit of two data subsets
[ 50]     call lsq_fit (j,            x(1:j),      y(1:j),      fit1)
[ 51]     call lsq_fit (lines-j, x(j+1:lines), y(j+1:lines), fit2)
[ 52]     error = fit1(3) + fit2(3)
[ 53]
```

In splitting up the two data regions not that it was not necessary to copy segments of the independent and dependent data. Instead the colon operator, or implied do loops, were used in lines 50 and 51 to pass vectors with j and $(lines - j)$ entries, respectively to the two calls to `lsq_fit`. After combining the two errors, in line 52, we update the current best choice for the data set division point in lines 55 through 58.

```
[ 54] !   does this division gives you a smaller error ?
[ 55]   if ( error < error_min ) then
[ 56]     error_min = error ; split = j
[ 57]     left      = fit1  ; right = fit2
[ 58]   end if ! current best choice
[ 59] end do ! of split choices
```

After we exit the loop, at line 59, we simply list the best results obtained. In line 73 we have also deallocated the data arrays even though it is just a formality at this point since all memory is released at the program terminates immediately afterwards. Had this been a subroutine or function then we would need to be sure that allocated variables are released when their access scope has terminated. Later versions of Fortran will do that for you, but good programmers should keep up with memory allocations.

```
[ 60] !   Display the results
[ 61]   print *, "Two line best fit; combined error is ", error_min
[ 62]   print *, "Best division of the data is:"
[ 63]   print *, "data(:j), data(j+1:), where j = ", split
[ 64]   print *, "Left line fit:"
[ 65]   print *, "the slope is      ", left(1)
[ 66]   print *, "the intercept is ", left(2)
[ 67]   print *, "the error is      ", left(3)
[ 68]   print *, "Right line fit:"
[ 69]   print *, "the slope is      ", right(1)
[ 70]   print *, "the intercept is ", right(2)
[ 71]   print *, "the error is      ", right(3)
[ 72]
[ 73]   deallocate (y, x)
[ 74] end program two_line_lsq_fit
[ 75]
```

For completeness an input routine, `read_xy_file`, is illustrated. It is elementary since it does not check for any read errors, and thus does not allow for any exception control if the read somehow fails.

```
[ 76] subroutine read_xy_file (infile, lines, x, y)
[ 77] !-----
[ 78] ! Take a file number, the number of lines to be read,
[ 79] !   and put the data into the arrays x and y
[ 80] !-----
[ 81]   implicit none
[ 82]   integer, intent(in) :: inFile    ! unit to read
[ 83]   integer, intent(in) :: lines    ! length of the file
[ 84]   real,    intent(out) :: x(lines) ! independent data
[ 85]   real,    intent(out) :: y(lines) ! dependent data
[ 86]   integer j
[ 87]   rewind (inFile) ! go to front of the file
[ 88]   do j = 1, lines ! for the entire file
[ 89]     read (infile, *) x(j), y(j) ! get the x and y values
[ 90]   end do ! over all lines
[ 91] end subroutine read_xy_file
[ 92]
```

If the supplied data file was huge, say argument `lines` has a value of ten million, the such data would probably have been stored in a binary rather than a formatted file. In that case we would simply invoke a binary read by re-writing line 89 as

```
[ 89]     read (infile) x(j), y(j) ! binary read of x and y
```

Such a change would yield a much faster input, but would still be relatively slow due to being in the loop starting at line 88. To get the fastest possible input we would have had to have saved the binary data on the file such that all the x values were stored first, followed by all the corresponding y values. In that case, we avoid the loop and get the fastest possible input by replacing lines 88–90 with:

```
[ 88]      ! sequential binary read of x and y values
[ 89]      read (infile) x, y
[ 90]      ! input complete, add iostat for exceptions
```

Here we will not go into the details about how we would have to replace subroutine inputCount an equivalent one for binary files. To do that you will have to study the Fortran INQUIRE statement for files, and its IOLENGTH option to get a hardware independent record length of a real variable.

```
[ 93]      ! Given test data in file two_line.dat:
[ 94]      ! 0.0000000e+00  1.7348276e+01
[ 95]      ! 1.0000000e+00  6.5017349e+01
[ 96]      ! 2.0000000e+00  8.7237749e+01
[ 97]      ! 3.0000000e+00  1.2433478e+02
[ 98]      ! 4.0000000e+00  1.5456681e+02
[ 99]      ! 5.0000000e+00  1.8956219e+02
[100]      ! 6.0000000e+00  2.1740486e+02
[101]      ! 7.0000000e+00  2.3138619e+02
[102]      ! 8.0000000e+00  2.7995041e+02
[103]      ! 9.0000000e+00  3.1885162e+02
[104]      ! 1.0000000e+01  3.4628642e+02
[105]      ! 1.1000000e+01  3.3522546e+02
[106]      ! 1.2000000e+01  3.7626218e+02
[107]      ! 1.3000000e+01  3.9577060e+02
[108]      ! 1.4000000e+01  4.2217988e+02
[109]      ! 1.5000000e+01  4.3388828e+02
[110]      ! 1.6000000e+01  4.5897959e+02
[111]      ! 1.7000000e+01  4.9506511e+02
[112]      ! 1.8000000e+01  5.0747649e+02
[113]      ! 1.9000000e+01  5.2168101e+02
[114]      ! 2.0000000e+01  5.2976511e+02
```

Assuming the formatted data are stored in file two_line.dat, as shown above we obtain the best two straight line fit.

```
[115]      ! Running the program gives:
[116]      !
[117]      ! Enter the data input filename: two_line.dat
[118]      ! There were 21 records read.
[119]      ! Single line fit
[120]      ! the slope is      25.6630135
[121]      ! the intercept is  53.2859993
[122]      ! the error is      343.854675
[123]      ! Two line best fit; combined error is 126.096634
[124]      ! Best division of the data is:
[125]      ! data(:j), data(j+1:), where j = 11
[126]      ! Left line fit:
[127]      ! the slope is      31.9555302
[128]      ! the intercept is  24.9447269
[129]      ! the error is      46.060421
[130]      ! Right line fit:
[131]      ! the slope is      21.6427555
[132]      ! the intercept is 112.166664
[133]      ! the error is      80.0362091
[134]
```

Check this out by plotting the data points and the three straight line segments. Just remember that the first line covers the whole domain, while the second goes only up to halfway between points 11 and 12 while the third line runs from there to the end of the independent data.

C.12 Problem 4.11.15 : Find the next available file unit

The INQUIRE statement has a lot of very useful features that return information based on the unit number, or the file name. It can also tell you how much storage a particular type of record requires (like the sizeof function in C and C++). Here we use only the ability to determine if a unit number is currently open. To do that we begin by checking the unit number that follows the last one we utilized. Line 9 declares that variable, last_unit and initializes it to 0. The save attribute in that line assures that the latest value of last_unit will always be saved and available on each subsequent use of the function. Since the standard input/output units have numbers less than ten we allow the unit numbers to be used to range from 10 to 999, as seen in line 8. However, the upper limit could be changed.

Lines 14–18 determine if the unit after last_unit is closed. If so that unit will be used and we are basically finished. We set the return value, next, update last_unit, and return.

```
[ 1] function get_next_io_unit () result (next)
[ 2] ! * * * * *
[ 3] ! find a unit number available for i/o action
[ 4] ! * * * * *
[ 5] implicit none
[ 6] integer :: next ! the next available unit number
[ 7]
[ 8] integer, parameter :: min_unit = 10, max_unit = 999
[ 9] integer, save :: last_unit = 0 ! initialize
[10] integer :: count ! number of failures
[11] logical :: open ! file status
[12]
[13] count = 0 ; next = min_unit - 1
[14] if ( last_unit > 0 ) then ! check next in line
[15] next = last_unit + 1
[16] inquire (unit=next, opened=open)
[17] if ( .not. open ) last_unit = next ! found it
[18] return
```

Otherwise, if the unit after last_unit is open we must loop over all the higher unit numbers in search of one that is closed. If we succeed then we update last_unit and return by exiting the forever loop, as seen in lines 24 and 25.

```
[19] else ! loop through allowed units
[20] do ! forever
[21] next = next + 1
[22] inquire (unit=next, opened=open)
[23] if ( .not. open ) then
[24] last_unit = next ! found it
[25] exit ! the unit loop
[26] end if
```

At this point it may be impossible to find a unit. However, with 999 units available it is likely that one that was previously in use has now been closed and is available again. Before aborting we reset the search and allow three cycles to find a unit that is now free. That is done in lines 27–31.

```
[27] if ( next == max_unit ) then ! attempt reset 3 times
[28] last_unit = 0
[29] count = count + 1
[30] if ( count <= 3 ) next = min_unit - 1
[31] end if ! reset try
```

In the unlikely event that all allowed units are still in use we abort the function after giving some insight to why.

```
[32] if ( next > max_unit ) then ! abort
[33] print *, 'ERROR: max unit exceeded in get_next_io_unit'
[34] stop 'ERROR: max unit exceeded in get_next_io_unit'
[35] end if ! abort
[36] end do ! over unit numbers
[37] end if ! last_unit
[38] end function get_next_io_unit
```

C.13 Problem 5.4.4 : Polymorphic interface for the class 'Position_Angle'

```
[ 1] module class_Position_Angle ! file: class_Position_Angle.f90
[ 2] use class_Angle
[ 3] implicit none
[ 4] type Position_Angle ! angle in deg, min, sec
[ 5] private
[ 6] integer :: deg, min ! degrees, minutes
[ 7] real :: sec ! seconds
[ 8] character :: dir ! N | S, E | W
[ 9] end type
```

The above type definitions are unchanged. The only new part of the module for this class is the INTERFACE given in the following four lines.

```
[10] interface Position_Angle_ ! generic constructor
[11] module procedure Decimal_sec, Decimal_min
[12] module procedure Int_deg, Int_deg_min, Int_deg_min_sec
[13] end interface
[14] contains . . .
```

Returning to the original main program:

```

[ 1] program main
[ 2]   use class_Great_Arc
[ 3]   implicit none
[ 4]   type (Great_Arc)      :: arc
[ 5]   type (Global_Position) :: g1, g2
[ 6]   type (Position_Angle) :: a1, a2
[ 7]   type (Angle)         :: ang
[ 8]   real                  :: deg, rad

```

We simply replace all the previous constructor calls with the generic function `Position_Angle_` as shown on lines 8 through 17 below.

```

[ 9]   a1 = Position_Angle_ (10, 30, 0., "N") ! note decimal point
[10]   call List_Position_Angle (a1)
[11]   a1 = Position_Angle_ (10, 30, 0, "N")
[12]   call List_Position_Angle (a1)
[13]   a1 = Position_Angle_ (10, 30, "N")
[14]   call List_Position_Angle (a1)
[15]   a1 = Position_Angle_ (20, "N")
[16]   call List_Position_Angle (a1)
[17]   a2 = Position_Angle_ (30, 48, 0., "N")
[18]   call List_Position_Angle (a2)

```

C.14 Problem 6.4.1 : Using a function with the same name in two classes

```

[ 1] include 'class_X.f90'
[ 2] include 'class_Y.f90'
[ 3] program main ! modified from Fig. 4.6.2-3F
[ 4]   use class_Y, Y_f => f ! renamed in main
[ 5]   implicit none
[ 6]   type (X_) :: x, z ; type (Y_) :: y
[ 7]   x%a = 22 ! assigns 22 to the a defined in X
[ 8]   call X_f(x) ! invokes the f() defined in X
[ 9]   print *, "x%a = ", x%a ! lists the a defined in X
[10]   y%a = 44 ! assigns 44 to the a defined in Y
[11]   x%a = 66 ! assigns 66 to the a defined in X
[12]   call Y_f(y) ! invokes the f() defined in Y
[13]   call X_f(x) ! invokes the f() defined in X
[14]   print *, "y%a = ", y%a ! lists the a defined in X
[15]   print *, "x%a = ", x%a ! lists the a defined in X
[16]   z%a = y%a ! assign Y a to z in X
[17]   print *, "z%a = ", z%a ! lists the a defined in X
[18] end program main ! Running gives:
[19] ! X_ f() executing ! x%a = 22
[20] ! Y_ f() executing ! X_ f() executing
[21] ! y%a = 44 ! x%a = 66
[22] ! z%a = 44

```

C.15 Problem 6.4.3 : Revising the employee-manager classes

The changes are relatively simple. First we add two lines in the `Employee` class:

```

interface setData ! a polymorphic member
module procedure setDataE ; end interface

```

Then we change two other lines:

```

[ 8]   empl = setData ( "Burke", "John", 25.0 )
[14]   mgr = Manager_ ( "Kovacs", "Jan", 1200.0 ) ! constructor

```

The generic `setData` could not also contain `setDataM` because it has the same argument signature as `setDataE` and the compiler would not be able to tell which dynamic binding to select.